Chapter 11

Implementing an Assembler and a Linker Using C++ and Java

An assembler must handle

- Absolute address
- Signed numbers
- Unsigned numbers
- External label
- Internal (i.e., local) label
- Public label
- Null-terminated string
- String that is not null-terminated
- "Label + unsigned_number" expression
- "Label - unsigned_number" expression
- "*"
- "* + unsigned_number" expression
- "* − unsigned_number" expression
- **dup** modifier
- **equ** symbol

11.2.1 Specifications for a Simple Assembler

We will call our first assembler version **masv1**. It will work like **mas**, with the following exceptions:

- The command line format of **masv1** is

  masv1 *<infilename>*

  If *<infilename>* does not include an extension, then **masv1** adds ".mas" to it. **masv1** never prompts the user for any input.
- The output file name is always

  <infilename_less_extension>.mac
- **masv1** does not support list files, configuration files, the !-directive, and the &-directive.
- **masv1** treats all numbers as decimal.
- **masv1** is always case sensitive when processing labels.
- **masv1** does not support strings. Thus, none of the following statements are allowed:

```
        ldc   'A'
s1:     dw    "ABC"
s2:     dw    'ABC'
```

- **masv1** does not support comments or lines containing only a label. However, it allows completely blank lines.
- The operand field may contain only a label, or an integer with an optional leading sign.
- The only directive **masv1** supports is the **dw** directive.
- **masv1** supports only the instructions in the standard instruction set.
- **masv1** uses a symbol table that can hold at most 20 entries (this small upper limit makes it easy to test the code in **masv1** that handles symbol table overflow).
- **masv1** should correctly handle filenames that include "." (representing the current directory) or ".." (representing the parent directory).
- When started, masv1 should display the following message (insert your own name):

  masv1 written by . . .

masv1, like **mas**, should return an error code of 1 on any error, and 0, otherwise. **masv1** should detect the following errors:

- Invalid operation (an item in the operation field other than a valid instruction mnemonic or **dw**)
- Ill-formed label in the label field
- Ill-formed label in operand field

- Undefined label in the operand field
- Duplicate label (two or more lines starting with the same label)
- Address or operand out of range. The operands in `dw` statements and the values in the `x` and `y` fields of instructions have ranges as follows:

```
dw operand:      -32768 to 65535
x field value:    0 to 4095
y field value:    0 to 255
```

  For example, some *illegal* statements are

```
dw     66000
ld     4096
aloc   256
aloc   -1
```

- Symbol table overflow
- Program too big (i.e., bigger than 4096 words)
- Incorrect number of command-line arguments
- Cannot open input file
- Cannot open output file

Whenever **masv1** detects an error, it should display an error message and terminate. The error message should include the line number on which the error occurs, the input file name, the line itself, and a description of the error. Here, for example, is a typical error message:

```
ERROR on line 1 of aprog4.mas:
a?b:    ldc    5
Ill-formed label in label field
```

masv1 should not check if an instruction has the proper number of arguments. Missing operands should default to 0; extra operands should be ignored.

Assembler pseudocode—pass 1

FIGURE 11.1

```
Set location_counter to 0.
Open input file as a text file
Open output file as a binary file
// Pass 1
Loop until no more input
    +---
    |    Read one line from the input file.
    |    If the current line starts with a label,
    |        get the label and enter it and the current value
    |        of location_counter into the symbol table.
    |    Get the operand on the current line.
    |    If the operand is a label,
    |        output 'R' and the current value of
    |        location_counter to the output file.
    |    Add 1 to location_counter
    +---
Output 'T'
Reset input file to the beginning.
```

Pass 2

```
// Pass 2
Loop until no more input
    +---
    |    Read one line from the input file.
    |    Get the operation (a mnemonic or dw) from the current line.
    |    If the operation is a mnemonic,
    |        look up the opcode in opcode table and save it in opcode.
    |    Get the operand from the current line.
    |    If the operand is a label,
    |        look up its address in the symbol table and save
    |        it in operand_value.
    |    else                            // operand is a number
    |        convert the operand to binary
    |        and save it in operand_value.
    |    If the operation is a mnemonic,
    |        construct the machine language instruction from
    |        opcode and operand_value, and save in machine_word.
    |    else                            // operation is a dw
    |        copy operand_value to machine_word.
    |    Output machine_word.
    +---
Close files
```

Symbol table
How to implement?

FIGURE 11.2

mnemonic	opcode
"ld"	0
"st"	1
"add"	2
.	.
.	.
.	.

Relationship of opcode index (position of its mnemonic in an alphabetized table) and the opcode

FIGURE 11.3

Opcodes (hex)	Length	Index Range (decimal)	Formula
0 - E	4	0 - 14	index
F0 - FE	8	15 - 29	0xF0 + (index - 15)
FF0 - FFE	12	30 - 44	0xFF0 + (index - 30)
FFF0 - FFFF	16	45 - 60	0xFFF0 + (index - 45)

Computing an opcode from its table index

```
0xF0 + (index - 15)
```

For example, the index of `desp` is 22 decimal. Thus, its opcode is

```
0xF0 + (22 - 15) = F7 hex
```

Opcode table in Java

FIGURE 11.4

```
String opcode_table[] = {

        "ld",           // opcode 0
        "st",           // opcode 1
        "add",          // opcode 2
          .
          .
        "desp",         // opcode F7
        "????",         // opcode F8
        "????",         // opcode F9
          .
          .
        "????",         // opcode FFF4
        "uout",         // opcode FFF5
          .
          .
        "halt"          // opcode FFFF
};
```

Implement the symbol table using the SymbolTable class

FIGURE 11.5

```
class SymbolTable {

    private String symbol[];        // label array
    private short address[];        // address array
    private int index;              // index of next available slot

    public SymbolTable(int size)
    {
     // Constructor creates symbol and address arrays
     // and initializes index.
    }

    public void enter(String label, short address)
    {
     // Enters label/address into table.
    }

    public short search(String label)
    {
     // Returns address of label.
    }

}
```

Assembler should check for duplicate labels when making an entry into the symbol table.

FIGURE 11.6

```
1            ld     x
2            add    y
3            st     z
4            halt
5   x:       dw     2
6   x:       dw     3
7   z:       dw     0
```

Text files

Contain ASCII codes exclusively.

A “.mas” file is an example of a text file.

Binary files

- Do not contain ASCII codes exclusively.
- “.mob” and “.mac” files are binary files.
- No translation of end-of-line markers occurs during input or output of binary files.

FIGURE 11.7

Creates text file.

a)

```
import java.io.*;
class P1
{
    public static void main(String[] args) throws IOException {
       PrintWriter outStream =
              new PrintWriter(new FileOutputStream("outfile1"));
       short x = 10;
       outStream.print(x);    // contents of x converted to ASCII decimal
       outStream.close();
    }
}
```

b)

```
see Version x.x

      0:    3130                                         10

Input file = outfile1
List file  = outfile1.see
```

Creates binary file

c)

```
import java.io.*;
class P2
{
    public static void main(String[] args) throws IOException {
    DataOutputStream outStream =
        new DataOutputStream(new FileOutputStream("outfile2"));
    short x = 10;
    outStream.writeShort(x); // contents of x outputed as is
    outStream.writeByte('C');
    outStream.close();
    }
}
```

d)

see Version x.x

```
      0:     000A 43                                        ..C

Input file = outfile2
List file  = outfile2.see
```

Creates text file

FIGURE 11.8 a)

```
#include <fstream>
using namespace std;

int main() {
    ofstream outStream;
    outStream.open("outfile3");
    short x = 10;
    outStream << x;
    outStream.write((char *)&x, sizeof(x));
    outStream.close();
    return 0;
}
```

b)

```
                              extra byte inserted during output
see Version x.x
      0:     3130  0D0A  00                          10...

Input file = outfile3
List file  = outfile3.see
```

Creates binary file

c)

```
#include <fstream>
using namespace std;

int main() {
    ofstream outStream;
    outStream.open("outfile4", ios::binary);
    short x = 10;
    outStream.write((char *)&x, sizeof(x));
    outStream << 'C';
    outStream.close();
    return 0;
}
```

d)

```
see Version x.x

        0:   0A00 43

Input file = outfile4
List file  = outfile4.see
```

Binary input file in Java

```
DataInputStream inStream = new DataInputStream(
                                 new FileInputStream("infile1"));
byte b;
short x;
b = inStream.readByte();
x = inStream.readShort();
```

Binary input file in C++

In C++, create an input stream with **ifstream**, and read using its **read** function. For example, to read from the binary file **infile2**, use

Use on systems that do not
use single newline to terminate
text line.

```
ifstream inStream;
inStream.open("infile2", ios::binary);
char b;
short x;
inStream.read(&b, sizeof(b));
inStream.read((char *)&x, sizeof(x));
```

Reading the input text file

In Java we create an input stream for the input text file with

```
BufferedReader inStream = new Bufferedreader(
                          new FileReader(inFileName));
```

In C++ we use

```
ifstream inStream;
inStream.open(inFileName);
```

where **inFileName** is a string that contains the input file name. We can then read one line in Java with

```
buffer = inStream.readLine();
```

or in C++ with

```
inStream.getline(buffer, sizeof(buffer));
```

where **buffer** is a **String** variable in Java and a **char** array in C++.

Creating an R entry

During pass 1, as the assembler scans each line, it tests if the operand field contains a label. If it does, it outputs the appropriate R entry. To output the first field of an R entry (the letter 'R'), we can use

```
outStream.writeByte('R');
```

in Java, or

```
outStream << 'R';
```

in C++. Then to output the required address (the address of the current line), we can simply output the address in the location counter with

```
outStream.writeShort(location_counter)
```

in Java, or with

```
outStream.write((char *)&location_counter, 2);
```

in C++.

assemble method in **Assembler** class constructs machine word

FIGURE 11.10

```
class Assembler {

  private String opcodeTable[] = {
    "ld",      // opcode 0
    "st",      // opcode 1
    "add",     // opcode 2
      .
      .
      .
    }

    public short assemble(String mnemonic, String operand, SymbolTable s)
    {
      // Assembles machine word from mnemonic and operand using
      // the symbol and opcode tables.
    }

}
```

Constructing the machine word requires shifting. For example, for a 4-bit opcode:

```
Machine_word = (short)(opcode <<12) | operand_value;
```

11.2.7 Writing Machine Text to the Output File

In pass 2, the assembler again scans each line of the source program. For each line, the assembler constructs the machine word into an instruction or a data word using the information in the opcode and symbol tables. Then to output the machine word, use

```
outStream.writeShort(machine_word);
```

in Java, or

```
outStream.write((char *)&machine_word, 2);
```

in C++, where **machine_word** is the variable containing the constructed machine word.

To handle endian problem (e.g., if assembler written in Java and linker in C++ on a PC).

FIGURE 11.11

```
short reverseOrder(short x)
{
    int y = ((int) x) & 0xffff;              // promote with no sign ext
    return (short) (256 * (y%256) + y/256);
}
```

Using reverseOrder

Use

```
outStream.writeShort(reverseOrder(machine_word));
```

instead of

```
outStream.writeShort(machine_word);
```

11.3.1 Specifications for a Simple Linker

Our first version of a linker will be a simplified version of **lin** called **linv1**. It will work like **lin** with the following exceptions:

- The format of the command line is

  linv1 *<infilename> <infilename>* . . .

  If *<infilename>* does not include an extension, **linv1** adds ".mob" to it. At least one *<infilename>* must be specified. **linv1** never prompts the user for any input.
- The output file name is always

  <first_infilename_less_extension>.mac

- **linv1** is always case sensitive when processing labels.
- **linv1** does not support linking with libraries.
- **linv1** does not support list, answer, table, or cat files.
- **linv1** uses P, E, and R tables that can hold at most five entries (this small upper limit makes it easy to test the code in **linv1** that handles P, E, and R table overflow).
- `linv1` should correctly handle filenames that include "." (representing the current directory) or ".." (representing the parent directory).
- When started, `linv1` should display the following message (insert your own name):

  ```
  linv1 written by . . .
  ```

linv1, like **lin**, should return an error code of 1 on any error, and 0, otherwise. If any of the following errors occur during a link, **linv1** should generate an error message and terminate:

- Unresolved external symbol
- Duplicate public symbol
- More than one starting address

Linker should detect these errors

- Linked program too large (greater than 4096 words)
- Unlinkable input file (i.e., a file without a valid header or text)
- P, E, or R table overflow
- Incorrect number of command-line arguments
- Cannot open input file
- Cannot open output file

Error messages should include specific information on the error whenever possible. For example, if an external symbol is unresolved, the error message should indicate which external symbol.

As linker loads modules, it keeps track of its module's starting address

The linker determines the address (relative to the beginning of the machine code text) of the object module it is processing by using the relationship

address of current object module

=

address of previous object module

+

size of the text in the previous object module

and stores it in a variable named `module_address`. The address of the first module, of course, is 0.

Implement the P, E, R, and S tables using classes.

FIGURE 11.13 a)

```
class P    // class for the P table
{
    private String symbol[];
    private short address[];
    private int index;    //index of next available slot
    private static final int maxSize = 5;

    public P()
    {
        // constructor
    }

    public int search(String s)
    {
        // finds symbol in table and returns its index
    }

    public short getAddress(int i)
    {
        // returns address of entry at index i
    }

    public String getSymbol(index i)
    {
        // returns symbol of entry at index i
    }

    public int size()
    {

        // returns current size of table
    }

    public void enter(short add, String sym)
    {

        // enters new symbol and address
    }

    public void write(DataOutputStream s)
    {
        // writes out table
    }
}
```

(continued)

Implement the text buffer using a class.

FIGURE 11.13 (continued) b)

```
class T   // class for the text buffer
{
   private final int mainMemorySize = 4096;
   private short buffer[];
   private int index;

   public T
   {
      // constructor
   }

   public void add(short x)
   {
      // add word to buffer
   }

   public void relocate(int address, int change)
   {
      // add change to word at address
   }

   public void write(DataOutputStream) throws IOException
   {
      // writes out text
   }
}
```

Linker pseudocode—phase 1

FIGURE 11.14

```
// Phase 1
Open output file as a binary file
Set module_address to 0
Clear text_buffer
For each file on the command line
    +---
    |   Open file as a binary file
    |   Read file into file_buffer.
    |   Loop
    |       +---
    |       |   Get header entry
    |       |   If T entry
    |       |       Move text into text_buffer following the
    |       |       text that is already there
    |       |       Set module_address to
    |       |           module_address +  size (in words) of text
    |       |       Break  from loop
    |       |   If big-S entry
    |       |       Insert 'S' and address into S table
    |       |   If small-s entry
    |       |       Insert 's' and address + module_address into S table
    |       |   If P entry
    |       |       Insert symbol and address + module_address into P table
    |       |   If E entry
    |       |       Insert symbol and address + module_address into E table
    |       |   If R entry
    |       |       Insert address + module_address and module_address
    |       |           into R table
    |       +---
    +---
```

```
// Phase 2
Process each E entry
Process each R entry

// Phase 3
Output P table entries as P header entries
Output R table entries as R header entries
Output E table entries as R header entries
Output S table entry as s/S header entry
Output 'T'
Output text_buffer
Close files
```

Processing modules from a library

1. Reads in the module
2. Appends its text to the text already in the text buffer
3. Adds its P, E, R, and s/S entries with appropriately adjusted addresses to the P table, E table, R table, and S table
4. Resolves the external reference